Degree Thesis, Åland University of Applied Sciences, Bachelor of Information Technology

# DEVELOPMENT OF A SPORTS MANAGEMENT API

Jan Jakobsson

2023:33

Approval date: 16.11.2023
Supervisors: Agneta Eriksson-Granskog and Björn-Erik Zetterman

# EXAMENSARBETE
# Högskolan på Åland



| | |
|---|---|
| **Utbildningsprogram:** | Informationsteknik |
| **Författare:** | Jan Jakobsson |
| **Arbetets namn:** | Utveckling av maskingränssnitt för hantering av idrottsaktiviteter |
| **Handledare:** | Agneta Eriksson-Granskog, Björn-Erik Zetterman |
| **Uppdragsgivare:** | IFK Mariehamn Ishockey r.f. |





| **Abstrakt** |
|---|
| Syftet med examensuppdraget var att utveckla ett API med REST-tjänster som kan användas för att administrera ishockeyligor och presentera statistik.<br><br>Applikationen är utvecklad i Java med ramverket Quarkus, och kompilerad med GraalVM.<br><br>Resultatet blev en containeriserad applikation med REST-tjänster som kan distribueras och köras inom olika plattformar. Behörighet för de olika tjänsterna styrs genom rollbaserad åtkomstkontroll. |





| **Nyckelord (sökord)** |
|---|
| Java, Quarkus, GraalVM, Cloud, Container, Authentication |

| **Högskolans serienummer:** | **ISSN:** | **Språk:** | **Sidantal:** |
|---|---|---|---|
| 2023:33 | 1458-1531 | Engelska | 35 sidor |

| **Inlämningsdatum:** | **Presentationsdatum:** | **Datum för godkännande:** |
|---|---|---|
| 07.09.2023 | 13.05.2020 | 16.11.2023 |

# DEGREE THESIS
# Åland University of Applied Sciences


| Study program: | Information Technology |
| --- | --- |
| Author: | Jan Jakobsson |
| Title: | Development of a Sports Management API |
| Academic Supervisor: | Agneta Eriksson-Granskog, Björn-Erik Zetterman |
| Technical Supervisor: | IFK Mariehamn Ishockey r.f. |



**Abstract**

The purpose of this thesis is to develop an API with REST services that can be used to administer ice hockey leagues and aggregate statistics.

The application is developed in Java with the Quarkus framework, and compiled with GraalVM.
Technologies used include JAX-RS, Docker, JSON Web Token and OpenID Connect.

The result is a containerized application providing REST services which can be distributed and deployed in different platforms. Authorization is restricted with role-based access control.



**Keywords**

Java, Quarkus, GraalVM, Cloud, Container, Authentication

| Serial number: | ISSN: | Language: | Number of pages: |
| --- | --- | --- | --- |
| 2023:33 | 1458-1531 | English | 35 pages |

| Handed in: | Date of presentation: | Approved on: |
| --- | --- | --- |
| 07.09.2023 | 13.05.2020 | 16.11.2023 |

# TABLE OF CONTENTS


**1. INTRODUCTION** **6**
- 1.1 Purpose 6
- 1.2 Method 6
- 1.3 Scope and Limitation 7

**2. CLOUD-BASED MICROSERVICE** **8**
- 2.1 Cloud computing 8
  - 2.1.1 IaaS 8
  - 2.1.2 PaaS 9
  - 2.1.3 SaaS 9
- 2.2 Monolithic vs. Microservices Architecture 10
- 2.3 Containers 11
  - 2.3.1 Docker 12
  - 2.3.2 Kubernetes 12
- 2.4 Quarkus 13
  - 2.4.1 Container-first 13
  - 2.4.2 Live Coding 14
- 2.5 GraalVM 14
  - 2.5.1 Dynamic (Just-In-Time, JIT) Compiler 15
  - 2.5.2 Static (Ahead-Of-Time, AOT) Compiler 15

**3. REST API** **17**
- 3.1 REST 17
  - 3.1.1 Client/Server separation 17
  - 3.1.2 JAX-RS 17
  - 3.1.3 JSON 18
- 3.2 Setup 18
- 3.3 Entity 18
- 3.4 Model 19
  - 3.4.1 Lombok 19
  - 3.4.2 MapStruct 21
- 3.5 Service 22
- 3.6 Endpoint 22
- 3.7 OpenAPI 24

**4. AUTHENTICATION** **25**
- 4.1 JSON Web Token 25
  - 4.1.1 Token Structure 26
  - 4.1.2 Token Header 26
  - 4.1.3 Token Payload 26
  - 4.1.4 Token Signature 27

4.1.5 JSON Web Encryption 27
4.2 OAuth 2.0 28
4.3 OpenID Connect 28
4.4 Role-Based Access Control 30
**5. CONCLUSION 31**
5.1 Result 31
5.2 Reflections 31
**REFERENCE LIST 32**

# 1. INTRODUCTION

This thesis describes the method, software concepts, and technology behind the construction of a cloud-based microservice with role-based access control.

## 1.1 Purpose

This project will be made for the benefit of IFK Mariehamn Ishockey r.f. where the purpose is to create an administration tool to manage a local hockey league. Required player information should be imported from an external source they already use to register their member base.

The requirements of the systems are:
The administrator should be able to

1. import players from an external source
2. set up leagues
3. generate match schedules
4. manage match results

The end user should be able to display (read-only) tabular data for league, team, and player.

## 1.2 Method

The application was developed using evolutionary prototyping (Floyd, 1984). The development lifecycle consists of design, implementation, and evaluation as the prototype is incrementally transitioning to a product. This is a dynamic approach where not all requirements must be known from the beginning.

Out of 5 possible Java (Oracle, 2020) frameworks, two proof of concepts were made for the REST API: One in Spring Boot (Spring, 2020) and the other in Quarkus (Quarkus, 2020a). The concepts also included fetching data from an external REST API. Quarkus was then chosen for its lightweight container-first approach.

The official documentation and guides provided most of the information needed for me to learn more about Quarkus.

## 1.3 Scope and Limitation

The application will use Docker to package the executable in a Docker image, which is then deployed through Kubernetes. The project will not go into detail on how to configure Docker or Kubernetes.

# 2. CLOUD-BASED MICROSERVICE

This chapter describes the concepts of cloud computing, containerization of software, and how it all can be utilized by Quarkus, the Java framework chosen for this project.

## 2.1 Cloud computing

| On-Premises | IaaS<br>Infrastructure as Service | PaaS<br>Platform as Service | SaaS<br>Software as Service |
|---|---|---|---|
| Applications | Applications | Applications | Applications |
| Data | Data | Data | Data |
| Runtime | Runtime | Runtime | Runtime |
| Middleware | Middleware | Middleware | Middleware |
| O/S | O/S | O/S | O/S |
| Virtualization | Virtualization | Virtualization | Virtualization |
| Servers | Servers | Servers | Servers |
| Storage | Storage | Storage | Storage |
| Networking | Networking | Networking | Networking |

| Self-managed | Provider-supplied |
|---|---|

*Figure 1. Cloud computing service illustration.*

Cloud computing is the on-demand availability of computer system resources, as an alternative to on-premises. The foremost benefits are the cost-effective and scalable use of remote storage and computing power. Early concepts of multiple users sharing a computer system's resources date back to the 1960s when the time-sharing concept was developed. (Wikipedia, 2020b).

### 2.1.1 IaaS

Infrastructure as a Service (IaaS) provides an abstraction to manage low-level details of underlying network infrastructure like physical resources, location, data partitioning, scaling, security, backup, etc. (Wikipedia, 2020a). This is a flexible way to reduce the upfront cost and complexity of managing physical infrastructure (Microsoft Azure, 2023a).

### 2.1.2 PaaS

Platform as Service (PaaS) provides a platform for customers to develop, run, and manage applications without the complexity of building and maintaining the typical infrastructure involved in developing applications (Wikipedia, 2020c).

### 2.1.3 SaaS

Software as a Service (SaaS) is a software licensing and delivery model where software is provided by a subscription and is centrally hosted. With application logic running in the cloud a SaaS application may be utilized from a web browser on any device (Wikipedia, 2020d).

## 2.2 Monolithic vs. Microservices Architecture

A typical monolithic application is a single deployable with multiple business functions built from one codebase. Components in a monolithic architecture are tightly coupled and interdependent since data access is done directly within the same codebase (Amazon Web Services, Inc., 2023). During the early phase of development, it might be beneficial with a monolithic design since it is easier to build, deploy, and debug. As time grows the codebase of the monolith can be cumbersome to scale and maintain (Lavann, 2019). The Domain Driven Design (DDD) approach could help to maintain and evolve a large and complex codebase by abstracting a domain model with an explicitly bounded context (Evans, 2014).

Microservices architecture is an approach that breaks down software into small autonomous services as a type of distributed system. Each microservice is specialized to perform a single function or business logic with a well-defined interface (Amazon Web Services, Inc., 2023). The Domain Driven Design approach has great synergies with microservices architecture since the bounded context naturally helps defining a new microservice from scratch or when migrating from a monolithic application (Lavann, 2019).

## 2.3 Containers

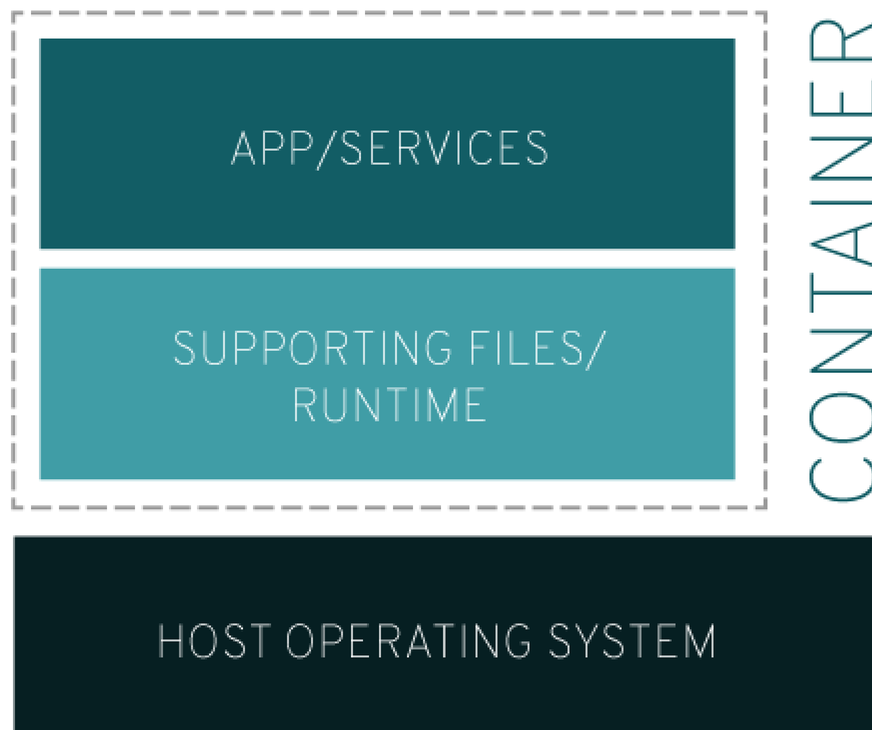


*Figure 2. Illustration of a container that is isolated from the system (Red Hat, 2020a).*

Unix V7 in 1979 introduced the chroot(2) system call, which was the first step of process isolation. It was not until the year 2000 that FreeBSD (FreeBSD Project, 2000a) released an OS-level virtualization mechanism called Jails based on the chroot concept. (FreeBSD Project, 2020b). Jails enable the computer system to be divided into several smaller systems, separate from each other. The principle of such containers are technologies that allow one to isolate applications into software packages that contain their entire runtime environment together with all necessary files needed to run (Red Hat, 2020a).

Prior to Java 11 the Java Virtual Machine was not fully container-aware (Schad, 2018), which could lead to the kernel shutting down a container as it attempted to fetch system resources outside its boundaries. This has then been improved through each major Java release (Soto, 2020).

Quarkus has extensions that provide support for several container image technologies (Quarkus, 2020b). Docker was chosen amongst them for this project.

### 2.3.1 Docker

Docker is a containerization technology that uses OS-level virtualization to deliver executable software packages, more known as Docker Images (Docker, 2023), which one can treat like lightweight and modular virtual machines. This gives flexible containers one can create, deploy, copy, and move between environments with ease. Some key advantages are modularity, image version control, ability to rollback, and rapid deployment (Red Hat, 2020b).

### 2.3.2 Kubernetes

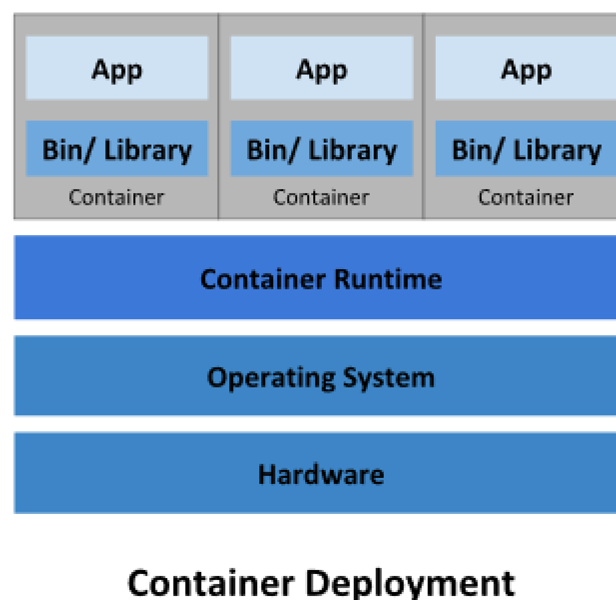


*Figure 3. Kubernetes is a platform to manage containerized applications (Kubernetes, 2020).*

Kubernetes is an open-source container orchestration platform that automates many of the manual steps involved when deploying, managing, and scaling containerized applications, like Docker. Kubernetes is an ideal platform for hosting cloud-native applications clusters as it can span hosts across on-premise, public, private, or hybrid clouds.

Google (the company Google/Alphabet) originally open-sourced Kubernetes in June 2014. The design and development for Kubernetes are derived from its predecessor, The Google Borg system (Kubernetes, 2020). As one of the early contributors to Linux container technology, Google has implied full use of containers as it was mentioned that Google runs everything in containers (Beda, 2014).

## 2.4 Quarkus

Quarkus is a Kubernetes Native Java framework tailored for GraalVM and HotSpot.
It is designed to work with popular standards, frameworks, and libraries like Eclipse MicroProfile, Spring, Apache Kafka, JAX-RS, Hibernate ORM, and many more.
For developers, Quarkus offers some ease with the Live Coding feature which will transparently compile any changed files upon receiving an HTTP request. It also provides unified imperative and reactive programming with an embedded managed event bus (Quarkus, 2020a).

### 2.4.1 Container-first

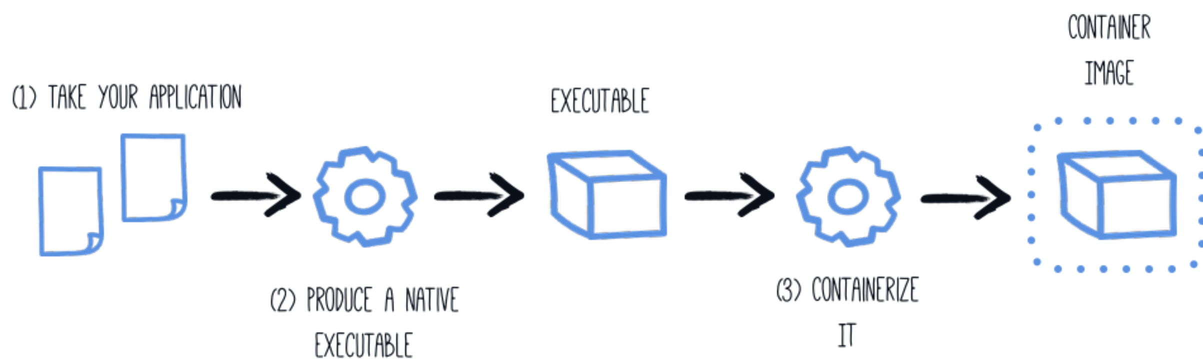


*Figure 4. Quarkus is designed with containers in mind (Quarkus, 2020c).*

Quarkus has been designed with a container-first philosophy. In real terms, what this means is optimization for low memory usage and fast startup times, such as:

- First-class support for Graal/SubstrateVM
- Build Time Metadata Processing
- Reduction in Reflection Usage
- Native Image Pre-Boot

### 2.4.2 Live Coding

When starting Quarkus in development mode the *Live Coding* feature is active. Most Java developers are familiar with a flow such as:

*Write Code → Compile → Deploy → Refresh Browser → Repeat*

With Live Coding active Quarkus will block any incoming HTTP request to check if any source files have been modified. If that is the case the modified files will be compiled and redeployed in the application with the new files:

*Write Code → Refresh Browser → Repeat*

With quick iterations, the development flow feels more productive (Quarkus, 2020c).

## 2.5 GraalVM

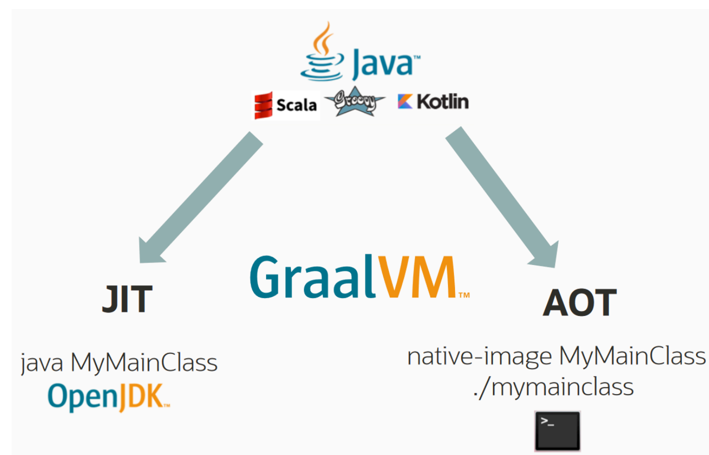


*Figure 5. GraalVM is a versatile Java VM and JDK (Wuerthinger, 2019a, 2019b).*

GraalVM is a universal virtual machine for running applications written in various languages. It also provides the ability to compile JVM bytecode to a native executable. This native executable runs a special virtual machine called SubstrateVM. When an application is compiled down to a native image it starts much faster and can run with a much smaller heap than a standard JVM (GraalVM, 2020b). GraalVM also has support for polyglot programming which allows one to write applications that can pass values from one programming language to another (GraalVM, 2020c).

### 2.5.1 Dynamic (Just-In-Time, JIT) Compiler

```
package jdk.vm.ci.runtime;

import jdk.vm.ci.code.CompilationRequest;
import jdk.vm.ci.code.CompilationRequestResult;

public interface JVMCICompiler {
    int INVOCATION_ENTRY_BCI = -1;

    /**
     * Services a compilation request. This object should compile the
     * method to machine code and install it in the code cache if the
     * compilation is successful.
     */
    CompilationRequestResult compileMethod(CompilationRequest request);
}
```

*Figure 6. The functional interface for the JVMCI Compiler.*

GraalVM JIT compiler is written completely in Java by using the JVM Compiler Interface (JVMCI) introduced in Java 9, and later backported to Java 8. This means it is easier for a Java developer to browse, debug and modify the GraalVM source code compared to a JIT compiler written in C/C++ like *javac*. When using a JIT compiler, the Java source code is compiled into a binary representation of the application, so called JVM bytecode. To run the application, the JVM needs to interpret the JVM bytecode and then compile it into machine code during runtime. This means the final code can be optimized depending on the workload (Seaton, 2017).

### 2.5.2 Static (Ahead-Of-Time, AOT) Compiler

The AOT compiler performs static analysis to create a single statically linked executable, called a native image. During this process, all unreachable code is discarded. Classes must be explicitly registered for reflection. The native image is highly optimized with instant startup and a small memory footprint (GraalVM, 2020b).

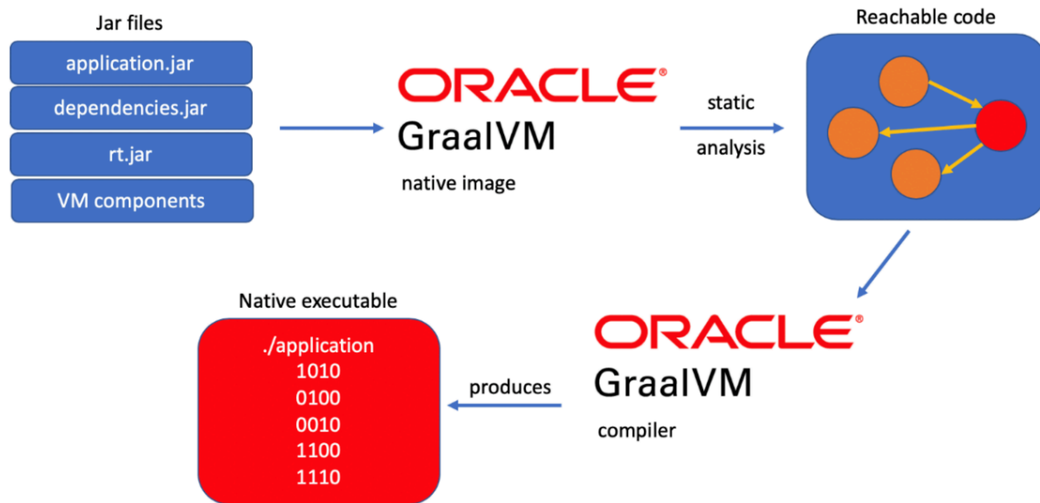


*Figure 7. AOT compilation time can be huge due to the static analysis (Delabassee, 2020a, 2020b).*

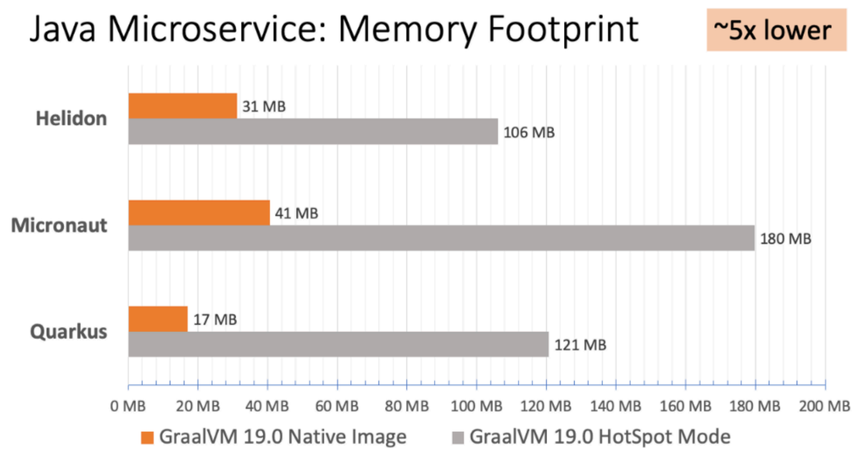


*Figure 8. Memory footprint with Java-based microservice frameworks (GraalVM, 2020a).*

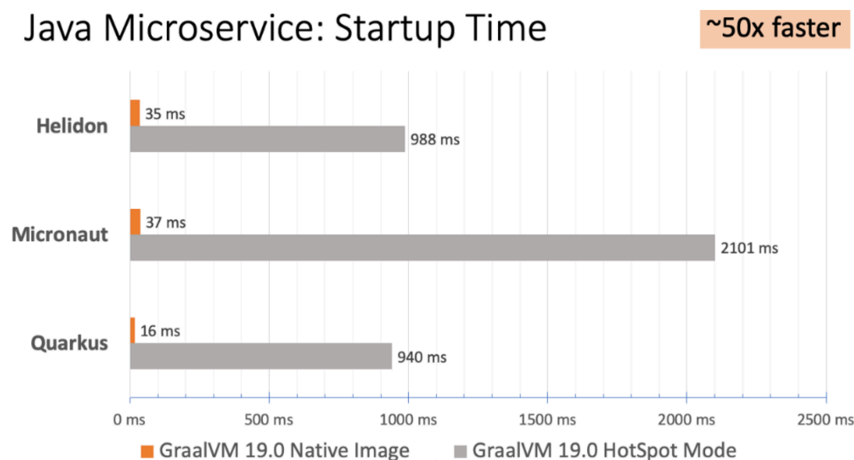


*Figure 9. Startup time with Java-based microservice frameworks (GraalVM, 2020a).*

# 3. REST API

An application programming interface (API) is an abstraction that defines how to build and integrate software. For the purpose of building a headless web application, a REST API was defined.

## 3.1 REST

Representational State Transfer (REST) is a software architectural style that defines standards between computer systems. REST systems are characterized by being stateless and separating the concerns of the client and the server.

A REST API uses HTTP methodologies defined by the RFC 2616 protocol (Internet Engineering Task Force, 1999). The most typical HTTP methods are:

- GET – retrieve a collection of resources or specific resources by identifier
- POST- create a new resource
- PUT – update specific resources by identifier
- DELETE – remove specific resource

### 3.1.1 Client/Server separation

REST is stateless by design. Implementation of the client and the implementation of the server can be done independently without each knowing about the other. In real terms, this means that the client-side code can change at any time without affecting the server-side code and vice versa. If the message format is known for both sides, they can be kept modular and separate. This improves scalability and flexibility.

### 3.1.2 JAX-RS

Quarkus utilizes Jakarta RESTful Web Services (JAX-RS) and provides a powerful API to expose RESTful web services by using declarative annotations (Eclipse Foundation, 2017).

### 3.1.3 JSON

Javascript Object Notation (JSON), defined in RFC 8259 (Internet Engineering Task Force, 2017), is an open standard data-interchange format derived from Javascript. JSON is a language-independent specification using conventions familiar to programming languages of the C-family. Most modern programming languages have support to generate and parse JSON data, which makes it suitable for RESTful web services (JSON, 2020).

## 3.2 Setup

A new Quarkus project was created by importing necessary dependencies with Gradle. A local Docker container with PostgreSQL was started for persistence. Quarkus was started in development mode and the coding could commence with small iterations.

## 3.3 Entity

```
@Entity
@Cacheable
public class League extends PanacheEntity {

    public String name;

    public LeagueType type;

    public Season season;

    @Column(name = "season_start", columnDefinition = "TIMESTAMP")
    public LocalDateTime seasonStart;

    @Column(name = "season_end", columnDefinition = "TIMESTAMP")
    public LocalDateTime seasonEnd;
}
```

*Figure 10. A simple entity with standard JPA annotations.*

Entities were created using Panache, a Hibernate ORM extension for Quarkus. By declaring fields public, they will get auto-generated accessors, like Lombok. The class must also be annotated as an Entity.

## 3.4 Model

During the first iterations of the application endpoints would invoke the Panache entities directly and serialize them into JSON. Simple data classes were introduced to decouple entity classes and avoid sending complex objects to the presentation layer. Data Transfer Objects (DTO) is one definition of such objects. They also give better control over the data which is exposed.

### 3.4.1 Lombok

Lombok was used to reduce repetitive coding in data classes. Instead of manually writing the typical methods for getters, setters, equality, and string representation the class can be annotated with *@Data*. The result is a class that is very similar to Kotlin's data class (Kotlin, 2020).

```
@Data
public class TeamDTO {

    protected Long id;

    protected String teamName;
}
```

*Figure 11. Example class using Lombok @Data annotation*

Lombok Annotation Processor will manipulate the class during compile time by delegating a handler that can modify the Abstract Syntax Tree. This allows Lombok to inject new nodes as methods. After processing is done the compiler will generate byte code from the modified AST (Project Lombok, 2020).

```java
public class TeamDTO {

    protected Long id;

    protected String teamName;

    public TeamDTO() {
    }

    public Long getId() {
        return this.id;
    }

    public String getTeamName() {
        return this.teamName;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public void setTeamName(@NotNull String teamName) {
        this.teamName = teamName;
    }

    public boolean equals(final Object o) {
        /** code omitted for brevity */
    }

    public int hashCode() {
        /** code omitted for brevity */
    }

    public String toString() {
        /** code omitted for brevity */
    }
}
```

*Figure 12. Class after deLombok*

If, for some reason, there is a need to abandon Lombok it is possible to generate the code with the *Delombok* feature.

### 3.4.2 MapStruct

```
@Mapper(config = QuarkusMapperConfig.class)
public interface PlayerMapper {

    PlayerDTO toPlayerDTO(Player player);

    Player toEntity(PlayerDTO playerDTO);

    PlayerDetailsDTO toPlayerDetailsDTO(Player player);

    Player toEntity(PlayerDetailsDTO playerDetailsDTO);
}
```

*Figure 13. A Mapper interface that can be injected into the service layer*

MapStruct was used to ease up mapping between Entity and DTO. It was chosen due to the fact it generates code at compile time, which makes it just as fast as any hand-written mapper. If no additional configuration is needed one just needs to define a simple interface for MapStruct to generate the code needed for mapping,

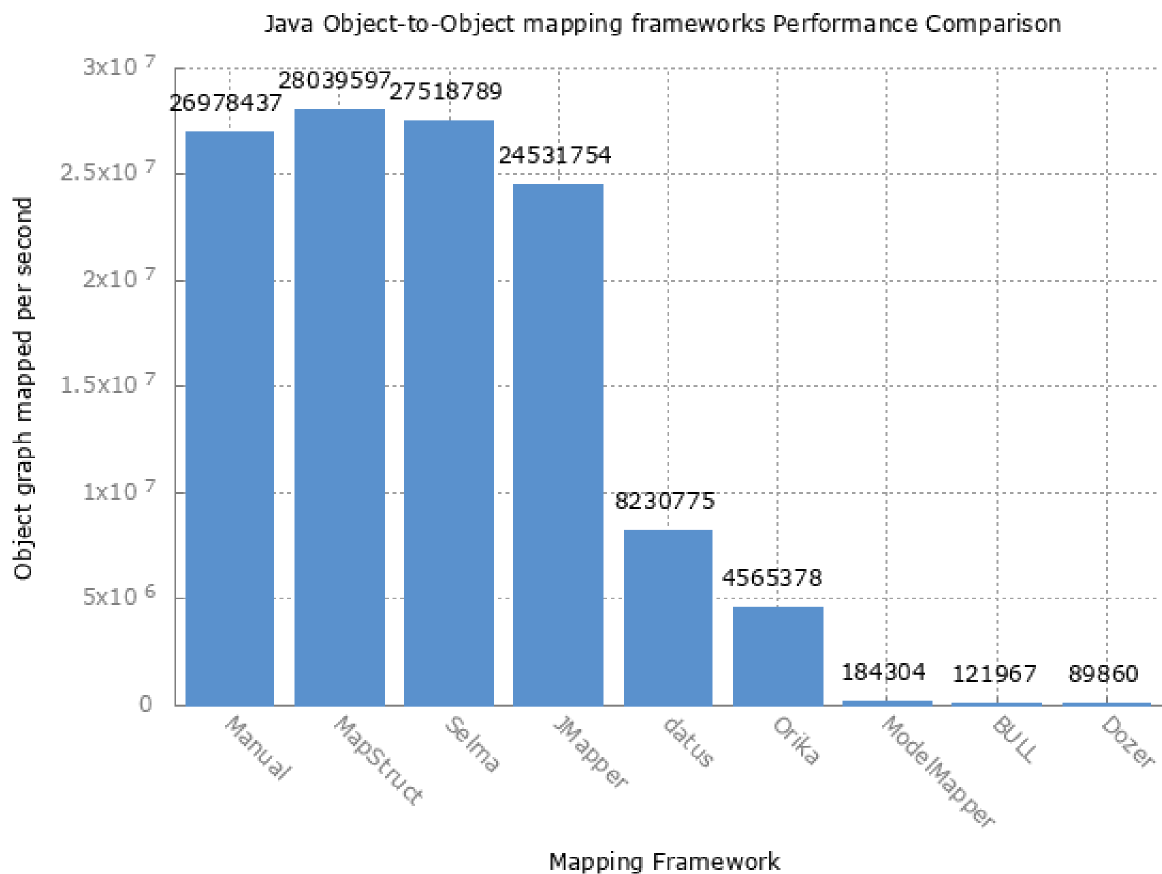


*Figure 14. Benchmark results show MapStruct does well compared to other mapping frameworks (Rey, 2015).*

## 3.5 Service

To further keep the Entity classes simple, a Service layer was implemented to handle the mapping between Entity and DTO.

```
@ApplicationScoped
@Transactional(REQUIRED)
public class PlayerService {

    @Inject
    PlayerMapper playerMapper;

    @Transactional(SUPPORTS)
    public List<PlayerDTO> findAllPlayers() {
        Stream<Player> players = Player.streamAll();
        return players.map(player -> playerMapper.toPlayerDTO(player))
                .collect(Collectors.toList());
    }
```

*Figure 15. Panache provides clean syntax to fetch a list of all players*

## 3.6 Endpoint

Endpoints based on entities declared. Quarkus utilizes Jakarta RESTful Web Services (JAX-RS), which provides a powerful API to expose RESTful web services by using declarative annotations (Eclipse Foundation, 2017).

```java
@Path("/players")
@Produces(APPLICATION_JSON)
@ApplicationScoped
public class PlayerResource {

    @GET
    public Response all() {
        /** invoke player service*/
    }

    @GET
    @Path("{id}")
    public Response player(@PathParam Long id) {
        /** invoke player service */
    }


    @POST
    public Response addPlayer(@Valid PlayerDetailsDTO player) {
        /** invoke player service */
    }

    @PUT
    public Response updatePlayer(@Valid PlayerDetailsDTO player) {
        /** invoke player service */
    }

    @DELETE
    public void removePlayer(){
        /** invoke player service */
    }
```

*Figure 16. Player endpoint using JAX-RS annotations.*

```
$ curl -s GET "http://localhost:80/players/1" -H "accept application/json"

{
  "id": 1,
  "firstName": "Hattrik",
  "lastName": "Swayze",
  "jerseyNumber": 11,
  "position": "FORWARD",
  "active": true,
  "currentTeam": {
    "id": 1,
    "teamName": "Solar Bears"
  }
}
```

*Figure 17. JSON response from endpoint after invoking request with curl command.*

### 3.7 OpenAPI

```
@Tag(ref = "Player", description = "Player operations")
@Path("/players")
@Produces(APPLICATION_JSON)
@ApplicationScoped
public class PlayerResource {

    @Inject
    PlayerService playerService;

    @Operation(summary = "Returns a player for a given identifier")
    @APIResponse(
        responseCode = "200",
        content = @Content(
            mediaType = APPLICATION_JSON,
            schema = @Schema(implementation = PlayerDetailsDTO.class)))
    @APIResponse(
         responseCode = "204",
         description = "The player is not found for a given identifier")
    @GET
    @Path("{id}")
    public Response player(
        @Parameter(description = "Player identifier", required = true)
        @PathParam Long id) {

        PlayerDTO player = playerService.findPlayerDetailsById(id);

        if (player != null) {
        return Response.ok(player).build();
        } else {
        return Response.noContent().build();
        }
    }
```

*Figure 18. Player endpoint with Swagger annotations for documentation and API definition.*

Quarkus' OpenAPI extension generates a YAML-based definition for OpenAPI Specification (OAS), originally known as Swagger Specification. The OAS defines a language-agnostic interface to RESTful APIs which allows clients to discover and consume the service without access to source code or knowledge about the server implementation.

The declarative specification can later be used to automatically generate model objects for the frontend implementation, which is outside the scope of this project (Swagger, 2020).

# 4. AUTHENTICATION

To secure admin operations authentication is a must. Traditionally this is done by stateful, session-based architecture, where a user would provide credentials, and generate a unique session ID which would be stored on the server and sent back to the user. This requires the server to create and store the session in the data store.

One approach to stateless authentication is by token-based authentication using the Authorization HTTP header defined in RFC 7235 (Internet Engineering Task Force, 2014).

```
{
  "Authorization": "Bearer $TOKEN"
}
```

*Figure 19. Authorization header.*

## 4.1 JSON Web Token


eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4


*Figure 20. Example JWT is generated at jwt.io with the HS256 signing algorithm.*

JSON Web Token (JWT) is an open standard, RFC 7519 (Internet Engineering Task Force, 2015), for creating data with optional signature and/or optional encryption whose payload holds JSON that asserts some number of claims. The JWT is a compact and URL-safe way of representing claims between sender and recipient. The claims are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure.

### 4.1.1 Token Structure

*Figure 21. JWT has 3 sections (Gulati, 2018).*

The token consists of three sections delimited by two periods:

1. Header
2. Payload
3. Cryptographic signature

### 4.1.2 Token Header

The header is Base64Url encoded and tells us the type of token and hashing algorithm used. HMAC SHA 256 and RSA are two common algorithms used.

```
/** Header */
{
  "typ": "JWT",
  "alg": "HS256"
}
```

*Figure 22. Token header*

### 4.1.3 Token Payload

```
/** Payload */
{
  "sub": "1234567890",
  "name": "John Doe",
  "jti": "777603b8-f450-46bf-8f24-c2ad6300287a",
  "iat": 1590440100,
  "exp": 1590443810,
  "groups": "Everyone"
}
```

*Figure 23. Token payload.*

The payload holds a set of claims. There are three types of claims defined in the RFC 7519 specification, none of which are mandatory.
Registered claims: a set of useful and interoperable claims like *sub*, *iss* and *exp*.
Public claims: custom claim names defined at will.
Custom claims: custom claims producer and consumer may agree to use.

In the above example:

- "sub" (Subject) identifies the user. A registered claim.
- "name" Custom private claim.
- "jti" (JWT ID) provides a unique id for the JWT. A registered claim.
- "iat" (Issued At) Claim identifies the time the JWT was issued. A registered claim.
- "exp" (Expiration Time) Claim identifies the JWT expiration time. A registered claim
- "groups" Custom private claim for Role-Based Access Control.

### 4.1.4 Token Signature

The signature is the last part of the JWT where the encoded header, encoded payload, and a secret are added to the hashing algorithm. By using the signature, the provider and consumer can verify that the received token is unaltered.
When a JWT is signed it is referred to as JSON Web Signature (JWS). It is important to understand that encoding is different from encryption. The claims in a JWS are still public.

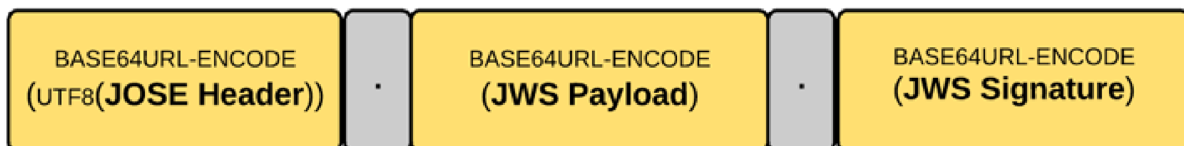


*Figure 24. Compact representation of JWS (Siriwardena, 2016).*

### 4.1.5 JSON Web Encryption

Encrypted JWT is referred to as JSON Web Encryption (JWE). It is often nested by first creating a JWS which then is encrypted to JWE.

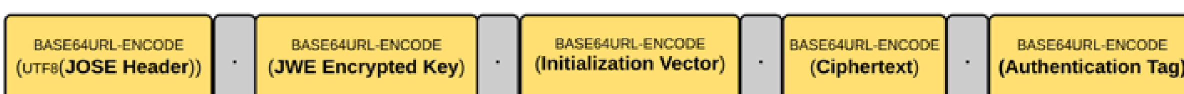


*Figure 25. Compact representation of JWE (Siriwardena, 2016).*

## 4.2 OAuth 2.0

The OAuth 2.0 authorization protocol is used to control the scope for Role-Based User-Access. OAuth2.0 provides specific authorization flows for web applications, desktop applications, mobile phones, and smart devices. For this project, Okta's SaaS platform was chosen to provide identity and access management (Okta, 2020b) instead of hosting an authorization service like Keycloak (Keycloak, 2020).

## 4.3 OpenID Connect

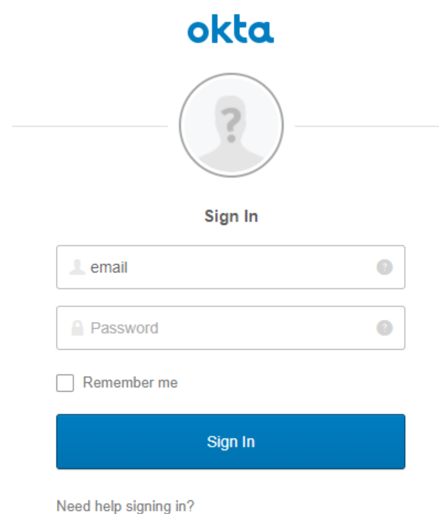


*Figure 26. The default Okta OIDC authentication prompt, as seen from the application under development.*

OpenID Connect (OIDC) extends Oauth 2.0. OIDC provides user authentication and single sign-on (SSO) functionality. Quarkus OIDC extension gives a seamless and user-friendly flow when a user is requesting a secured resource.

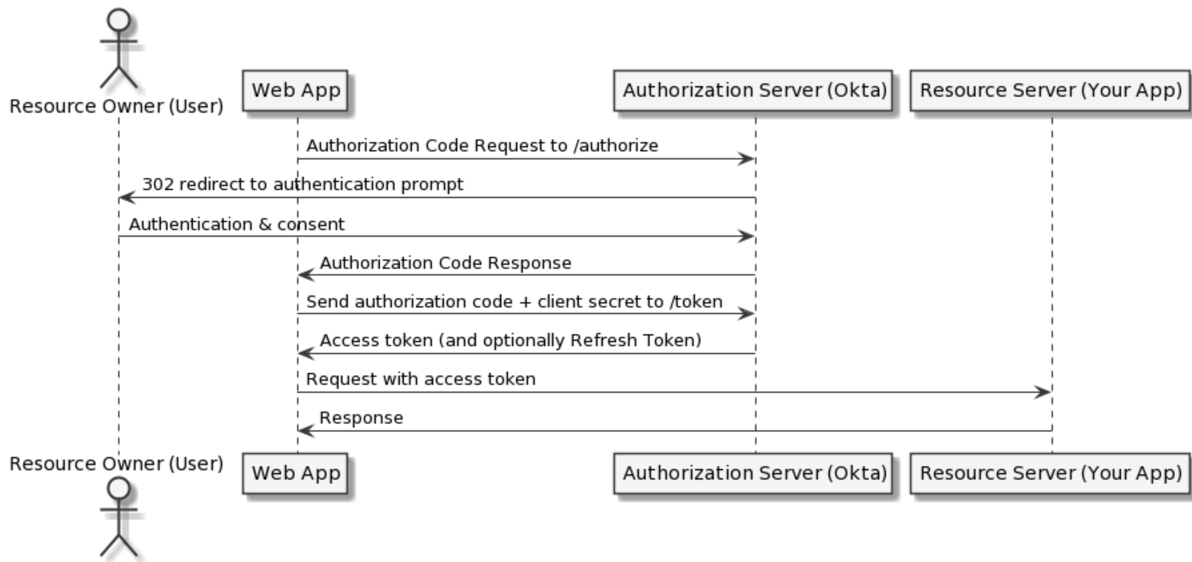


*Figure 27. Authorization Code Flow (Okta, 2020).*

An unauthenticated user requesting a secured resource is redirected to the authorization prompt. Once granted, the authorization code can be exchanged for an access token (JWT), and optionally refresh token. The secured resource can now be accessed by validating the JWT. Quarkus OIDC extension does this by storing the access token in a session cookie.

To minimize the risk of Cross-Site Request Forgery (CSRF) (OWASP Foundation, 2020) a state cookie is created before the redirect to the authorization prompt. The state cookie is then matched against the state query parameter in the callback from the identity provider (Okta, 2020b).

## 4.4 Role-Based Access Control

Role-Based Access Control (RBAC) is an approach to manage permissions to users based on their role within the organization, which is easier to control compared to individual permissions. Quarkus OIDC extension for RBAC utilizes Eclipse MicroProfile library to provide secured access to endpoints.

```
@RequestScoped
@Path("/secured")
public class JwtSecuredResource {

    @Inject
    JsonWebToken jwt;

    @GET
    @RolesAllowed("admin")
    @Produces(MediaType.TEXT_PLAIN)
    public String admin(@Context SecurityContext securityContext) {
        return securityContext.getUserPrincipal().getName();
    }

    @GET
    @PermitAll
    @Produces(MediaType.TEXT_PLAIN)
    public String everyone(@Context SecurityContext ctx) {
        Principal caller =  ctx.getUserPrincipal();
        return caller == null ? "anonymous" : caller.getName();
    }
```

*Figure 28. Example of how a resource can be secured with JWT.*

# 5. CONCLUSION

## 5.1 Result

The application was deployed in Azure Kubernetes Service at no cost, made possible by applying for a Microsoft Azure Sponsorship for nonprofit organizations. A front-end application is still required to utilize the API properly for end users.

My goal of making the base for a REST API with Role-Based Access Control with a new framework surpassed my expectations. I already had some experience with token-based authentication prior to this project, which is why I was very pleased with how seamless it was to integrate with an external identity provider. It is just one example of how Quarkus aims for, as they call it, developer joy. The drawback of a fresh framework with a smaller community is the knowledge base when you run into problems, in addition to the learning curve involved.

In the future, I would like to extend my knowledge with reactive programming and streams like Apache Kafka.

## 5.2 Reflections

My reflections are added well over three years after finishing the project described in this thesis. Migrating to a microservice architecture seems to be a business decision more companies are willing to invest in. Perhaps the step is easier to take when microservices can be broken out by reusing code from a monolithic codebase. This would be an easier task if the monolith was implemented with proper Domain Driven Design. Just containerizing the monolithic application in the cloud could help an organization tremendously to cut infrastructure costs.

# REFERENCE LIST


Beda, J. (2014, May 22). *Containers at scale - At Google, the Google Cloud Platform and Beyond*. Retrieved June 15, 2020, from Speakerdeck: https://speakerdeck.com/jbeda/containers-at-scale

Benevides, R. (2020, May 15). *Java inside docker: What you must know to not FAIL*. Retrieved from Red Hat Developer: https://developers.redhat.com/blog/2017/03/14/java-inside-docker/

Delabassee, D. (2020a, May 2-6). *Java in Containers - Part Deux* [Conference presentation]. QCon, London, England. https://www.infoq.com/presentations/openjdk-containers/

Delabassee, D. (2020b, May 2-6). *Java in Containers - Part Deux* [PDF slides]. QCon, London, England. https://archive.qconlondon.com/system/files/presentation-slides/david_delebassee_-_qcon_container_delabassee.pdf

Docker Inc. (2023, August 31). *Docker overview*. Retrieved August 31, 2023, from Docker Inc.: https://docs.docker.com/get-started/overview/)

Eclipse Foundation. (2017, August 7). *JAX-RS 2.1 API Specification*. Retrieved June 15, 2020, from JAX-RS: https://jax-rs.github.io/apidocs/2.1/

Evans, E. (2014). *Domain-Driven Design reference: Definitions and Pattern Summaries*. DogEar Publishing.

Floyd, C. (1984). *A Systematic Look at Prototyping*. In: Budde, R., Kuhlenkamp, K., Mathiassen, L., Züllighoven, H. (Eds.) *Approaches to Prototyping*, 1–18. Springer, Berlin, Heidelberg.

FreeBSD Project. (2000a, March 14). *FreeBSD 4.0 Release Notes*. Retrieved June 15, 2020, from FreeBSD Project: https://www.freebsd.org/releases/4.0R/notes/

FreeBSD Project. (2020b). *FreeBSD Handbook - Chapter 14. Jails*. Retrieved June 15, 2020, from FreeBSD Project: https://www.freebsd.org/doc/handbook/jails.html?3kwh

GraalVM. (2020a). *Why GraalVM?* Retrieved June 15, 2020, from GraalVM: https://www.graalvm.org/why-graalvm/

GraalVM. (2020b). *Native Image*. Retrieved June 15, 2020, from GraalVM: https://www.graalvm.org/reference-manual/native-image/

GraalVM. (2020c). *Write Polyglot Programs*. Retrieved June 15, 2020, from GraalVM: https://www.graalvm.org/reference-manual/polyglot-programming/

Gulati, R. (2020c). *Part 2: JWT to authenticate Servers API's*. Retrieved June 15, 2020, from Codeburst: https://codeburst.io/jwt-to-authenticate-servers-apis-c6e179aa8c4e

Internet Engineering Task Force. (1999, June). *Hypertext Transfer Protocol -- HTTP/1.1*. Retrieved June 15, 2020, from Internet Engineering Task Force: https://tools.ietf.org/html/rfc2616

Internet Engineering Task Force. (2014, June). *Hypertext Transfer Protocol (HTTP/1.1): Authentication - 4.2. Authorization*. Retrieved June 15, 2020, from https://tools.ietf.org/html/rfc7235#section-4.2

Internet Engineering Task Force. (2015, May). *JSON Web Token (JWT)*. Retrieved June 15, 2020, from Internet Engineering Task Force: https://tools.ietf.org/html/rfc7519

Internet Engineering Task Force. (2017, December). *The JavaScript Object Notation (JSON) Data Interchange Format*. Retrieved June 15, 2020, from Internet Engineering Task Force: https://tools.ietf.org/html/rfc8259

JSON. (2020). *Introducing JSON*. Retrieved June 15, 2020, from JSON: https://www.json.org/json-en.html

Kotlin. (2020). *Data Classes*. Retrieved June 15, 2020, from Kotlin: https://kotlinlang.org/docs/reference/data-classes.html

Kubernetes. (2020). *What is Kubernetes?* Retrieved June 15, 2020, from Kubernetes: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Lavann. (2019, September 4). *Monoliths to microservices using domain-driven design - Azure Architecture Center*. Microsoft Learn. Retrieved September 1, 2023, from https://learn.microsoft.com/en-us/azure/architecture/microservices/migrate-monolith

Okta. (2020). *OAuth 2.0 and OpenID Connect Overview*. Retrieved June 15, 2020, from Okta: https://developer.okta.com/docs/concepts/oauth-openid/

*Oracle*. (2020, June 15). Retrieved June 15, 2020, from Oracle Java: https://www.oracle.com/java/

OWASP Foundation. (2020). *Cross Site Request Forgery (CSRF)*. Retrieved June 15, 2020, from OWASP Foundation: https://owasp.org/www-community/attacks/csrf

Quarkus. (2020a, May 15). *Quarkus - Supersonic Subatomic Java*. Retrieved May 15, 2020, from Quarkus: https://quarkus.io/

Quarkus. (2020b). *Quarkus - Container Images*. Retrieved June 15, 2020, from Quarkus: https://quarkus.io/guides/container-image

Quarkus. (2020c). *Quarkus - Building a Native Executable*. Retrieved June 15, 2020, from Quarkus: https://quarkus.io/guides/building-native-image

Red Hat. (2020a). *What is a Linux container?* Retrieved June 15, 2020, from Red Hat: https://www.redhat.com/en/topics/containers/whats-a-linux-container

Red Hat. (2020b, June 15). *Containers - What is Docker?* Retrieved from Red Hat: https://www.redhat.com/en/topics/containers/what-is-docker

Rey, A. (2015, September 14). *Object-to-object mapping framework microbenchmark*. Retrieved June 15, 2020, from GitHub: https://github.com/arey/java-object-mapper-benchmark

Schad, J. (2018, January 2). *Nobody puts Java in a container*. Retrieved June 15, 2020, from JAXenter: https://jaxenter.com/nobody-puts-java-container-139373.html

Siriwardena, P. (2016, April 27). *JWT, JWS and JWE for Not So Dummies! (Part I)*. Retrieved 15 June, 2020, from Medium: https://medium.facilelogin.com/jwt-jws-and-jwe-for-not-so-dummies-b63310d201a3

Soto, A. (2020, May 15). *Quarkus – what's next for the lightweight Java framework?* Retrieved May 15, 2020, from JAXenter: https://jaxenter.com/quarkus-whats-next-for-the-lightweight-java-framework-160793.html

Spring. (2020, June 15). *Spring Boot*. Retrieved June 15, 2020, from Spring: https://spring.io/projects/spring-boot

Swagger. (2020, June 15). *OpenAPI Specification*. Retrieved June 15, 2020, from Swagger: https://swagger.io/specification/

Wikipedia. (2020a, June 1). *Infrastructure as a service*. Retrieved June 1, 2020, from Wikipedia: https://en.wikipedia.org/wiki/Infrastructure_as_a_service

Wikipedia. (2020b, June 1). *Cloud computing: Wikipedia.org*. Retrieved June 1, 2020, from Wikipedia: https://en.wikipedia.org/wiki/Cloud_computing

Wikipedia. (2020c, June 1). *Platform as a service*. Retrieved June 1, 2020, from Wikipedia: https://en.wikipedia.org/wiki/Platform_as_a_service

Wikipedia. (2020d, June 1). *Software as a service*. Retrieved June 1, 2020, from Wikipedia: https://en.wikipedia.org/wiki/Software_as_a_service

Wikipedia. (2020e). *Representational state transfer*. Retrieved June 15, 2020, from Wikipedia: https://en.wikipedia.org/wiki/Representational_state_transfer

Wuerthinger, T. (2019a, June 24-28). *Maximizing Performance with GraalVM* [Conference presentation]. QCon, New York, NY, United States. https://www.infoq.com/presentations/graalvm-performance/

Wuerthinger, T. (2019b, June 24-28). *Maximizing Performance with GraalVM* [PDF slides]. QCon, New York, NY, United States. https://archive.qconnewyork.com/system/files/presentation-slides/qcon2019.pdf